WHITE PAPER ARM®

Running ARM7TDMI® Processor Software on the Cortex™-M3 Processor

Mark Collier

November 2006

1 Introduction

The first question software developers will consider when looking at running ARM7TDMI processor software on a Cortex-M3 processor is: "How much time and effort is it going to take?" The answer is: “practically zero”. A simple re-targeting compilation will be all that is required in the vast majority of cases. Once the software is running, the developer may wish to consider enhancing it to exploit the new functionality of the Cortex-M3 processor not available on the ARM7TDMI processor. This whitepaper will explain how quickly and easily ARM7TDMI processor code can be ported to the Cortex-M3 processor, and how to use the additional features introduced with Cortex-M3 processor and the v7-M architecture to optimise software for size, power and maintainability.

1.1 Overview of Cortex-M3 Processor

The Cortex-M3 processor is the first ARM processor core based on the ARMv7-M architecture and has been specifically designed to achieve high system performance in power- and cost-sensitive deeply embedded applications, such as microcontrollers, automotive body systems, industrial control systems and wireless networking.

The Cortex-M3 processor has many attractive features compared to the ARM7TDMI processor:

- Architecture ARMv7-M with 36 new instructions since architecture ARMv4T.
- Higher performance and industry-leading code density with Thumb®-2 Instruction Set Architecture (ISA).
- Lower latency interrupts, standardised memory map, Memory Protection Unit, and integrated NVIC with built-in SysTick.
- Enhanced debug and trace capability.
- Greater power savings with two sleep modes.
- Better support for multi-core systems with CoreSight and other system features.

See the ‘Introduction to the Cortex-M3 processor’ white paper for a more substantial discussion of the Cortex-M3 processor features and benefits.

1.2 Getting Started

The majority of applications created for modern 32-Bit microcontrollers, such as the ARM7TDMI processor, are written in C or C++ to make code easily maintainable and portable between different families and vendors of devices. The majority of these applications will run on the Cortex-M3 processor with a simple re-targeting compilation.

Additionally, in most cases well-written application code will be abstracted from the underlying hardware through the use of Real-Time Operating Systems (RTOS)'s or low-level device drivers written by the microcontroller vendor. As such, if a user has application code written to run on a RTOS, such as uC-OS/II, Nucleus or ThreadX, running this on the Cortex-M3 processor may be as simple as installing the latest edition of the RTOS and targeting the new device. If the user is not running an RTOS then the application should run without source code changes, providing the microcontroller vendor has updated the low-level drivers which the application requires.

However, from the device vendors' perspective there is naturally some work to do to support any new device, whether it is an extension of an existing device or family, or the adoption of an extended architecture such as the Cortex-M3 processor. This white paper examines the easily identified source code and minor tool chain changes required to optimize low-level ARM7TDMI processor embedded software drivers for the Cortex-M3 device.

While modern developers predominantly use 'C', there are still cases where traditional assembler code may be used, for example in timing critical sections. If the assembler is written in 16-Bit Thumb code then this will often work "as is" with the Cortex-M3 processor as the native Thumb-2 ISA is a super-set of Thumb. However, even if 32-Bit ARM assembler has been used the assembler will be able to interpret these instructions as Thumb-2 instructions with little or no manual intervention. This will be covered in more depth in this white paper.

This paper assumes the reader is familiar with embedded software development and porting software between target devices.

1.3 ARM7TDMI Processor and Cortex-M3 Processor Comparison

The principal areas of difference between the ARM7TDMI processor and the Cortex-M3 processor from a software porting perspective are:

Feature	ARM7TDMI Processor	Cortex-M3 Processor
Execution state	ARM or Thumb	Thumb-2 only
Memory Map	Unstructured	Architecture Defined
Interrupts	IRQ / FIQ External interrupt controller	NMI + SysTick + up to 240 interrupts Integrated NVIC interrupt controller
System Status	PSR (Banked registers)	xPSR (Stacked registers)

Figure 1 Principal Software Differences between ARM7TDMI Processor and Cortex-M3 Processor

The ARM7TDMI processor may be either in ARM (32-bit instructions) or Thumb (16-bit instructions) state. The state is under software control, however, interrupt routines are always entered in ARM state. By contrast, the Cortex-M3 processor only executes in Thumb-2 state (16- and 32-bit instructions). See section 1.2 for more information on Thumb-2 technology.

The ARM7TDMI processor requires exception vectors to be located from address 0 onwards, but otherwise imposes no restrictions on where memory-mapped devices are located. By contrast, the Cortex-M3 processor imposes some order to improve portability between devices without sacrificing differentiation. However the memory-map regions were carefully chosen to fit with the majority of existing ARM7TDMI processor-based MCUs.

The ARM7TDMI processor has two external interrupts lines: IRQ (Interrupt Request) and FIQ (Fast Interrupt Request). Handling more than two interrupts requires an external interrupt controller such as a VIC (Vectored Interrupt Controller) or GIC (Generic Interrupt Controller) to multiplex the signals onto the IRQ (or FIQ) line. By contrast, the Cortex-M3 processor assumes many external interrupts will be required and so includes an integrated NVIC (Nested Vectored Interrupt Controller) which features an NMI (Non-Maskable Interrupt).

The ARM7TDMI processor has a complex programmer's model with six modes and 20 banked registers spread across the modes. By contrast, the Cortex-M3 processor is much simpler with only two modes and a single banked register. This means that registers are automatically stacked on an exception.

See section 3 for more information on the new features of Cortex-M3 processor which may be exploited after the application has been ported.

1.4 Why is the Cortex-M3 Processor Thumb-2 only?

Processors supporting Thumb mode still require ARM instructions in certain cases. Thumb-2 was designed to remove this restriction, permitting entire embedded software systems to be constructed using only Thumb-2 instructions. Therefore, the Cortex-M3 processor is Thumb-2 only because there is no longer a need for an ARM decoder.

More importantly, Thumb-2 was designed to deliver ARM performance levels at Thumb code densities. It achieves this seemingly impossible goal by adding new 16- and 32-bit instructions to the Thumb ISA to permit operations to be encoded more concisely, and by eliminating the need for state switching.

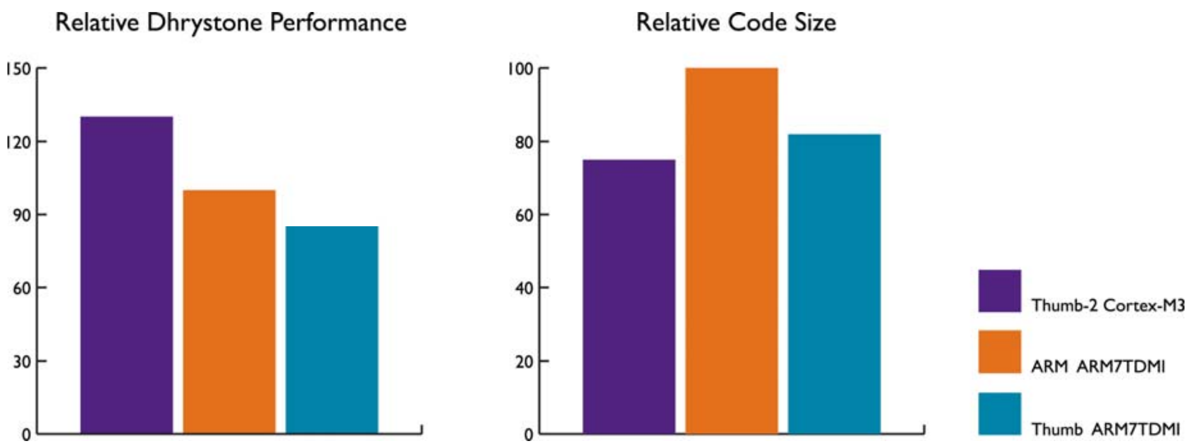

Figure 2 Relative Dhrystone and Code Sizes for Cortex-M3 Processor, and ARM7TDMI Processor ARM and Thumb State

Thumb-2 is a completely backward compatible with Thumb, so Thumb binaries can run unmodified. Thumb-2 is standard on all Cortex family cores.

2 Components of an Embedded Software System

The sections below assume that work has been performed to ensure that software accesses to the hardware are performed with the correct address and register definitions. This will be required when moving between any two target devices, irrespective of core, and will depend on the degree of similarity between the two devices in terms of clocks, interrupts, address map and programmer's interface of devices used.

2.1 Real-Time Operating System

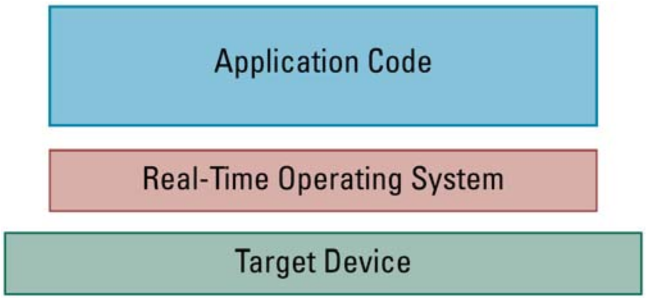

Figure 3 Components of an RTOS Embedded Software System

If the application uses a Real-Time Operating System (RTOS) then it is shielded from many details of the target device. Assuming the RTOS is available for Cortex-M3 processor, the application will need a simple re-targeting compilation. Developers may wish to re-build any Thumb objects to obtain the performance benefits of Thumb-2, however, this is not necessary for simple ports.

The assembler should have no difficulty assembling the ARM application assembler into Thumb-2 instructions due to the very high coverage of ARM functionality by Thumb-2; it just needs to know Cortex-M3 processor is the target. Developers may wish to re-write inline or embedded assembler included for performance reasons in more easily maintained C due to the higher efficiency of Cortex-M3 processor.

2.2 No Operating System

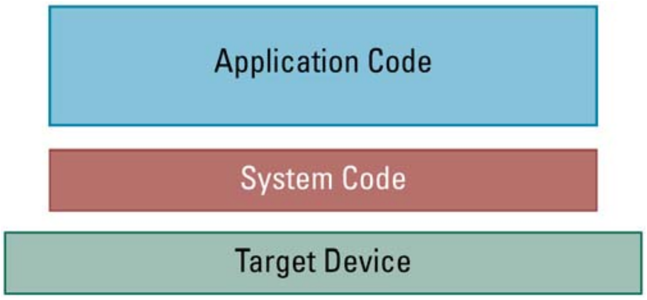

Figure 4 Components of an Embedded Software System

If the application does not make use of an RTOS, the code in the software system will normally be partitioned into Application Code and System Code.

2.2.1 Application Code

The Application Code will be the majority of the software and will need a simple re-targeting compilation. Developers may wish to re-build any Thumb objects to obtain the performance benefits of Thumb-2, however, this is not necessary for simple ports.

The assembler should have no difficulty assembling the ARM application assembler into Thumb-2 instructions due to the very high coverage of ARM functionality by Thumb-2; it just needs to know Cortex-M3 processor is the target. Developers may wish to re-write inline or embedded assembler included for performance reasons in more easily maintained C due to the higher efficiency of Cortex-M3 processor.

2.2.2 System Code

The System Code will be a minority of the software concerned with accesses to the underlying hardware and is normally supplied by the device vendor as a library and/or set of examples.

In the unlikely event, the device is not supplied with a library, the System Code can normally be written predominantly in C, which can be simply re-compiled as per Application Code. The small fraction remaining will be ARM assembler code to handle:

- Start-Up and Vector Table.
- Exceptions.
- Access to system features governed by instructions or core registers.

Typically the start-up code and exception vector table are contained in a single assembler source file. This file will be simpler for the Cortex-M3 processor than for the ARM7TDMI processor because the vector table is composed of addresses rather than ARM instructions, and there is no longer a requirement for code to initialise the banked registers in all six modes. Indeed, with RVDS it is possible to remove the assembler source file altogether and use the linker to set the reset vector and fix up the C stack and heap (further initialisation can then be performed by the C application).

Exception handling is easier with Cortex-M3 processor because registers are automatically stacked and un-stacked so exception handlers (including re-entrant ones) may be written exclusively in C and do not require interactions with external interrupt handlers, thereby saving code.

Very short functions providing access to system features, for example, to enable or disable interrupts may need porting. Typically there are only a handful of such functions.

Note: ARM assembler which uses the deprecated ‘mov pc’ instruction to return from functions will need these instructions replacing with ‘bx’ instructions.

3 Exploiting Cortex-M3 Processor

The previous sections have discussed how to take an existing application written for an ARM7TDMI processor target device and port it to a Cortex-M3 processor target device. The scope of the porting work will depend on the degree of similarity of the system design and the level of usage of ARM7TDMI features not present or handled differently in Cortex-M3 processor. This section aims to suggest areas when the newly ported application can be enhanced by taking advantage of the features the Cortex-M3 processor has which the ARM7TDMI processor does not have. RVCT 3.0 will automatically select from the 36 new instructions if they are more appropriate for expressing the desired behaviour.

3.1 Bit Manipulation

The Cortex-M3 processor has 6 new instructions to operate on bit fields within registers to help with peripheral programming and packet formation; to count leading zeros and to reverse bits in a register.

3.2 Byte and Half-word Manipulation

The Cortex-M3 processor has 3 new instructions to reverse bytes and half-words in a word and to unsigned and signed extend the Least-Significant Byte and Least-Significant Half-Word in a register to a 32-bit value.

3.3 Arithmetic

The Cortex-M3 processor has 11 new signed and unsigned hardware multiply and divide instructions to provide better support for saturation and 64-bit integer arithmetic.

3.4 Program Flow Control

The Cortex-M3 processor has 3 new instructions to enhance program flow control: ‘If Then’ to accommodate conditional execution; ‘Compare and branch if zero/no-zero’ for NULL pointer checks and ‘Table Branch’ for branch tables.

3.5 Multi-Processor Systems

The Cortex-M3 processor has 8 new instructions to improve operation in multi-processor systems. There are 3 new exclusive access instructions to manage semaphores shared between processors, 3 new memory barrier instructions for use when SRAM or external RAM is shared with another core, and 2 new instructions for sending events between processors.

3.6 Power Management

The Cortex-M3 processor has 2 new instructions which hint to the processor it can enter a low-power state until an interrupt or event occurs. These instructions are normally used in the idle thread. The Cortex-M3 processor can also be programmed to sleep whenever it has finished processing interrupts. Finally, the Cortex-M3 processor has a deep sleep mode which can be used if the target device supports it.

3.7 System Tick

The Cortex-M3 processor has a new 24-bit system timer integrated into the NVIC controller. This means RTOSs need not rely on external timers, thus making them significantly more portable.

3.8 Memory Protection Unit

The Cortex-M3 processor has an optional, integrated MPU which supports 8 memory regions of variable sizes. Attributes may be applied to the physical addresses of these regions to control the level of access permitted to user and privileged mode applications, and to describe the region (memory type, cache policy etc). Regions 256K or larger may be divided into 8 sub-regions to allow holes in the region where the attributes do not apply. Regions are allowed to overlap; the attributes of the highest region number defined dominate.

3.9 Debug

The Cortex-M3 processor has sophisticated built-in debug capabilities. Breakpoints, watchpoints, specified exceptions and external (off-chip/other core) debug requests can halt the core, or generate a ‘Hard Fault’ or ‘Debug Monitor’ exception. Halting the core allows it to be single-stepped. The ‘Debug Monitor’ exception allows a software monitor/status output to be invoked at each point. Core registers can also be read and written via externally accessible registers.

The ‘Flash Patch’ unit provides 2 literal comparators which can be used to substitute constants in the code region with values in SRAM or external memory; and 6 instruction comparators which can be used as breakpoint instructions, or to substitute instructions in the code region with values in SRAM or external memory.

The ‘Data Watchpoint and Trace’ unit provides 4 comparators which can be used as hardware watchpoints, Embedded Trace Module (ETM) triggers, PC or data address sampler event triggers. Watchpoints are events activated by a match to the PC address or type of data access to the data address which are similar to breakpoints in that they can halt the core or generate exceptions, but set a different flag in the Debug Fault Status Register. It also provides 6 counters for performance profiling, which can be used to emit ETM events on overflow.

The ‘Instrumentation Trace Macrocell’ provides a mechanism for the application to use the trace port to output debug messages with minimal overhead. It merges these packets with packets from the DWT, and can also inject delta timestamp packets into the stream.

In general, the application developer will not need to know all the details of the above because the features will be made available through the chosen debug tool.

3.10 Bit-Banding

The Cortex-M3 processor defines two bit-band regions in the address map:

- SRAM bit-band region for variables.
- Peripheral bit-band region for control and status registers.

By reading or writing a word from/to the bit-band alias address, individual bits can be accessed in the corresponding word in the bit-band region.

4 Case Study

To examine the above in practice a stand-alone C software application was written for the LuminaryMicro LM3S811 Cortex-M3 Evaluation Kit and ported to the Keil MCB2130 ARM7TDMI development board. The application allows a single LED to be switched on, off or blinked via a push-button, or text commands on the serial port. Interrupts are used extensively and divide and multiply are required to print the number of ticks since reset in decimal. Example assembler start-up files were used in both cases.

4.1 Code and Data Sizes

The code and data sizes (in bytes) for the application compiled (-O2) as ARM, ARM/Thumb and Thumb-2 are shown below:

Compiled Code and Data Section	ARM7TDMI ARM	ARM7TDMI ARM/Thumb	Cortex-M3 Thumb-2
Application Code	1596	1164	1000
Software Divide Code	364	364	0 (hardware divide)
Application Data	156	156	156
Start-up Code	272	272	200
Total	**2388**	**1956**	**1356**

Figure 5 Code and Data Sizes for a Functionally Equivalent Embedded Software System

4.2 Source Code Differences

Putting the two source C files for the ARM7TDMI processor and the Cortex-M3 processor side-by-side reveals the following differences (excluding white space and comments):

Source	ARM7TDMI Processor and Cortex-M3 Processor Source Differences
Application Source	Identical
Core System Source	13 lines different (3%)
Device System Source	122 lines different (29%)

Figure 6 Line Differences between Two Embedded Software Systems

The Core System Source is different because embedded assembler is used in the Cortex-M3 processor application to enable interrupts and take advantage of the new 'Wait For Interrupt' instruction (see 3.6). There is also a '#pragma thumb' in the ARM7TDMI processor application to compile for Thumb.

The Device System Source differs because the UART peripherals are different; the interrupt handling is by external VIC on the ARM7TDMI processor versus integrated NVIC on the Cortex-M3 processor; and the period timer interrupt is generated by external timer on the ARM7TDMI processor versus integrated System Tick on the Cortex-M3 processor.

The application used here is trivial; the system code percentages would be much smaller in a real embedded system.

5 Development Tools

Selecting the right development tools is a critical success factor in embedded software projects. This section examines the availability of tools for the Cortex-M3 processor.

5.1 ARM

ARM supplies a comprehensive range of development tools:

- RealView® Development Suite (RVDS) to enable the development of C, C++ and assembler software for the ARM architecture.
- RealView ICE and RealView Trace to connect RVDS to target devices for debug and trace.
- AMBA® Designer, RealView SoC Designer and RealView Model Library for hardware development.
- Instruction Set Simulation Models and Real-Time Simulation Models for s/w development.
- Emulation Baseboard for hardware FPGA prototyping.
- RealView Microcontroller Development Kit (Keil) to enable the development of embedded software on target devices and simulator, using the RealView Compilation Tools from RVDS.
- ULINK to connect RealView MDK to target devices for debug and trace.

See http://www.arm.com/products/DevTools for further information.

5.2 Ecosystem

There is a rich and growing ecosystem around the Cortex-M3 processor so while ARM supplies a complete end-to-end RealView tools solution, developers are increasingly able to remain with their tool chain of choice. The following development tools currently support the Cortex-M3 processor:

Tool	Supplier	Supplier Website
CrossWorks	Rowley Associates	www.rowley.co.uk
EDGE	Accelerated Technology	www.mentor.com
EWARM	IAR Systems	www.iar.com
GNU	CodeSourcery	www.codesourcery.com
TRACE32	Lauterbach	www.lauterbach.com

Figure 7 Cortex-M3 Processor Development Tools Vendors

The following RTOSs are currently available for the Cortex-M3 processor:

RTOS	Supplier	Supplier Website
FreeRTOS	FreeRTOS	www.freertos.org
Nucleus Plus	Accelerated Technology	www.mentor.com
ThreadX	Express Logic	www.rtos.com
RTX	ARM	www.arm.com
Salvo	Pumpkin	www.pumpkininc.com
uCOS-II	Micrium	www.micrium.com
uVelOSity	Green Hills Software	www.ghs.com

Figure 8 Cortex-M3 Processor RTOS Vendors

6 Conclusion

It has been seen that running ARM7TDMI processor application software on a Cortex-M3 processor device requires a simple re-targeting compilation and that running ARM7TDMI system software may require a small number of easily identifiable source code changes. The re-targeting compilation will automatically exploit the new v7-M instructions in the Cortex-M3 processor and in the case of RVDS select Thumb-2 libraries optimised for the Cortex-M3 processor.

Developers may choose to render operations formally written in assembler code in C, or to extend the system to take advantage of the new Cortex-M3 processor features. Developers have at their disposal a wide and growing range of tools and RTOSs to facilitate development, over and above the comprehensive support provided by ARM.

Switching the processor core of a system from the ARM7TDMI processor to the Cortex-M3 processor is a relatively simple task involving minimal effort over and above the normal work of porting between devices. Doing so will unlock the considerable competitive advantage of better performance, power consumption and higher code density that the Cortex-M3 processor provides.